Oid User's Manual

Robert J. Kingan and Sandra R. Kingan

1 August 2004

Contents

0.1 Introduction

Oid is an interactive extensible software system for studying matroids. Since matroids are a generalization of many other combinatorial objects such as graphs, matrices and linear spaces, a software system for matroid inherently handles all these objects. The name Oid comes from a humorous paper called "Oids and their Ilk" that gently makes fun of our tendency to work with the most abstract possible object.

Oid handles matroids representable over finite fields, as well as abstact matroids. The system deals flexibly with different matroid representations; most of its central algorithms work independently of the matroid is entered and stored in the system. It has, for example, an abstract isomorphism checker that can compare matroids entered as matrices with matroids entered as a family of basis sets. One of the main features of Oid is that it treats algorithms as data. Algorithms are plug-ins for Oid, so new algorithms can be added without recompiling the existing code.

Oid is written in the Java programming language. We selected Java because it is an object-oriented language with broad cross-platform support and a rich set of libraries for GUI elements, for which compilers and other development tools are freely available. While creating Oid we made the following design decisions that reflect our philosophy on computing:

1. Oid mirrors the structure of combinatorial objects. The object classes and algorithms correspond to combinatorial objects and algorithms. Moreover, the relationships between Oid's classes and algorithms correspond to the relationships between combinatorial objects and algorithms.

2. Oid treats algorithms as data. Data is not stored in a program's source code and can be manipulated by the program. Similarly, the names of algorithms are not hard-coded in the system. Instead, when Oid runs, it reads the list of algorithms it has available from a text file. Each algorithm is a Java class by itself so new algorithms can be easily added. Oid searches among the algorithms it has available to find the best one, in terms of complexity or user preference. New algorithms can be added without recompiling the system.

3. Oid is light with many small classes. We feel that a small system with an intuitive design is preferable to a large system without one.

4. Oid is interactive, but the classes in its library can be assembled into batch programs for larger computations. Oid's user-friendly GUI makes it suitable for small-scale experiments and pedagogical use. However, it is too cumbersome to work within the constraints enforced by the GUI for larger scale experiments and optimization

problems. In this case, the library of classes can be used to build customized batch programs. For example, the matroid generation program was built using Oid's class library [3].

5. Oid's design helps us maintain quality control as we add new features. By treating algorithms as plug-ins, the core library of code remains untouched.

A more detailed description of Oid's design goals can be found in [2].

Chapter 1 of this manual, "How to use Oid", provides step-by-step instructions for basic Oid tasks, including running Oid, entering new matroids, and loading and saving matroid. Chapter 2, "Matroid functions", covers the details of the matroid tasks available in Oid. Chapter 3, "Oid Concepts", covers the basic concepts behind Oid's design, including the structures that allow Oid to work with different representations, as well as Oid's internal representation of subsets and finite fields. Chapter 4, "Adding new algorithms to Oid" includes instructions for adding new algorithms to Oid.

Chapter 1

How to use Oid

1.1 Compiling and running Oid

Oid is written in Java. In order to run Oid, you must have the Java Runtime Environment installed on your machine, and it should be version 1.4.1_03 or later. The simplest way to check is to open a command prompt and enter the command:

```
java -version
```

If you have the correct version, something like this should appear:

```
java version "1.4.1_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1_03-b02)
Java HotSpot(TM) Client VM (build 1.4.1_03-b02, mixed mode)
```

If you need to install or upgrade your Java Runtime Environment, you can download a copy from http://java.sun.com/j2se/downloads/index.html.

In order to compile Oid from source code, you will also need a copy of the Java Software Development Kit (SDK). To check for this, type the command:

```
javac -help
```

If the Java SDK is installed, the system should respond with a list of `javac` command-line options. If you need a copy of the Java SDK, you can download it from http://java.sun.com/j2se/1.4.2/download.html.

The steps for compiling Oid on Windows or Unix/Linux platforms are very similar. The main difference is in the way subdirectories are specified. The instructions below assume

that you have extracted the files from oid.zip or oid.tar.gz into one directory, that you have a directory called `classes` to hold the compiled program, and that both directories are subdirectories of a common parent directory. You can, of course, place the source code in any directory you wish, and compile the program in any directory. Simply replace the directory names above with your own.

1.1.1 Unix/Linux

1. Create a destination directory for the compiled program, and copy the files:

   - `matroidTaskAlgorithmList.txt`
   - `matroidVisualizationList.txt`
   - `matroidVisTasks.txt`

   from the source directory to the destination directory.

2. Change to the source directory and type this command:

   ```
   javac -d ../classes @Oid/filelist.unix
   ```

   (Replace `../classes` with your own destination directory if its name is different.) This will compile the Oid library classes. The class files will be placed in a new subdirectory of your destination directory called `Oid`.

3. Next, type the command:

   ```
   javac -d ../classes @filelist
   ```

   Again, replace `../classes` with your own destination if it is different. This will compile the Oid program classes, and place the class files in your destination directory.

4. Finally, change to your destination directory and enter the following command to run Oid:

   ```
   java Oid
   ```

   If the compile succeeded, the Oid main window should appear.

1.1.2 Windows

The steps for compiling Oid for a Windows platform are quite similar to the Unix/Linux instructions:

1. Create a destination directory for the compiled program, and copy the files:

   - `matroidTaskAlgorithmList.txt`
   - `matroidVisualizationList.txt`
   - `matroidVisTasks.txt`

   from the source directory to the destination directory.

2. Change to the source directory and type this command:

   ```
   javac -d ..\classes @Oid\filelist.windows
   ```

   (Replace `..\classes` with your own destination directory if its name is different.) This will compile the Oid library classes. The class files will be placed in a new subdirectory of your destination directory called `Oid`.

3. Next, type the command:

   ```
   javac -d ..\classes @filelist
   ```

   Again, replace `..\classes` with your own destination if it is different. This will compile the Oid program classes, and place the class files in your destination directory.

4. Finally, change to your destination directory and enter the following command to run Oid:

   ```
   java Oid
   ```

   If the compile succeeded, the Oid main window should appear.

1.2 Oid's main window

Oid's main window, shown in Figure 1.1, is divided into three boxes. Since Oid can work with several matroids at the same time, the box on the left, called the *list box* displays a list of all the matroids loaded in the system. The box at the bottom is the *status box*. It provides status information on Oid's operations, that is, it tells you what Oid is doing at each step. Oid has has some rudimentary intelligence based on the complexity of the algorithms. If the smartmatroid class selects a different matroid representation to perform the task you selected, that information will show up in the status box. The third box is the *workspace box*. It displays all the matroids currently loaded and the results of computations. You can make the list box and the status box narrower, thereby increasing the workspace box.

In Figure 1.1, the user has entered the Fano matroid and clicked on "Generate dual in standard form" in the Tasks menu to produce the dual matroid. The user has also generated a list of the circuits of the Fano matroid, using the "Generate circuits" option, and the single-element extensions of the dual, using the "Single-element extensions of a GF(q) matroid by isomorphism" option.

1.3 Entering a new matroid

Oid handles both representable and non-representable matroids. Non-representable matroids can be entered via their family of bases. Rank 3 non-representable matroids can also be entered via their family of lines (rank 2 flats or hyperplanes). To enter a new matroid click "New" on the File menu. A dialog will appear with the following options:

- $GF(q)$**-representable matroids** prompts you to enter a matrix over a finite field to create a matroid.
- **Projective geometry** generates the projective geometry of a given rank over a finite field.
- **Basis representation** prompts you to enter a matroid by entering a list of its bases.
- **Linear space** prompts you to enter a linear space by a rank 3 matrix (if it is embeddable in a Desargusian projective plane) or by a list of its non-trivial lines.

If you selected **GF(q)-representable matroid** an entry window like the one in Figure 1.2 will appear. Enter the name, field size, size and rank of the new matroid, and then click the first "Continue" button. Then enter the values in the matrix. When the matrix is complete, click the second "Continue" button and the matroid will appear in its own

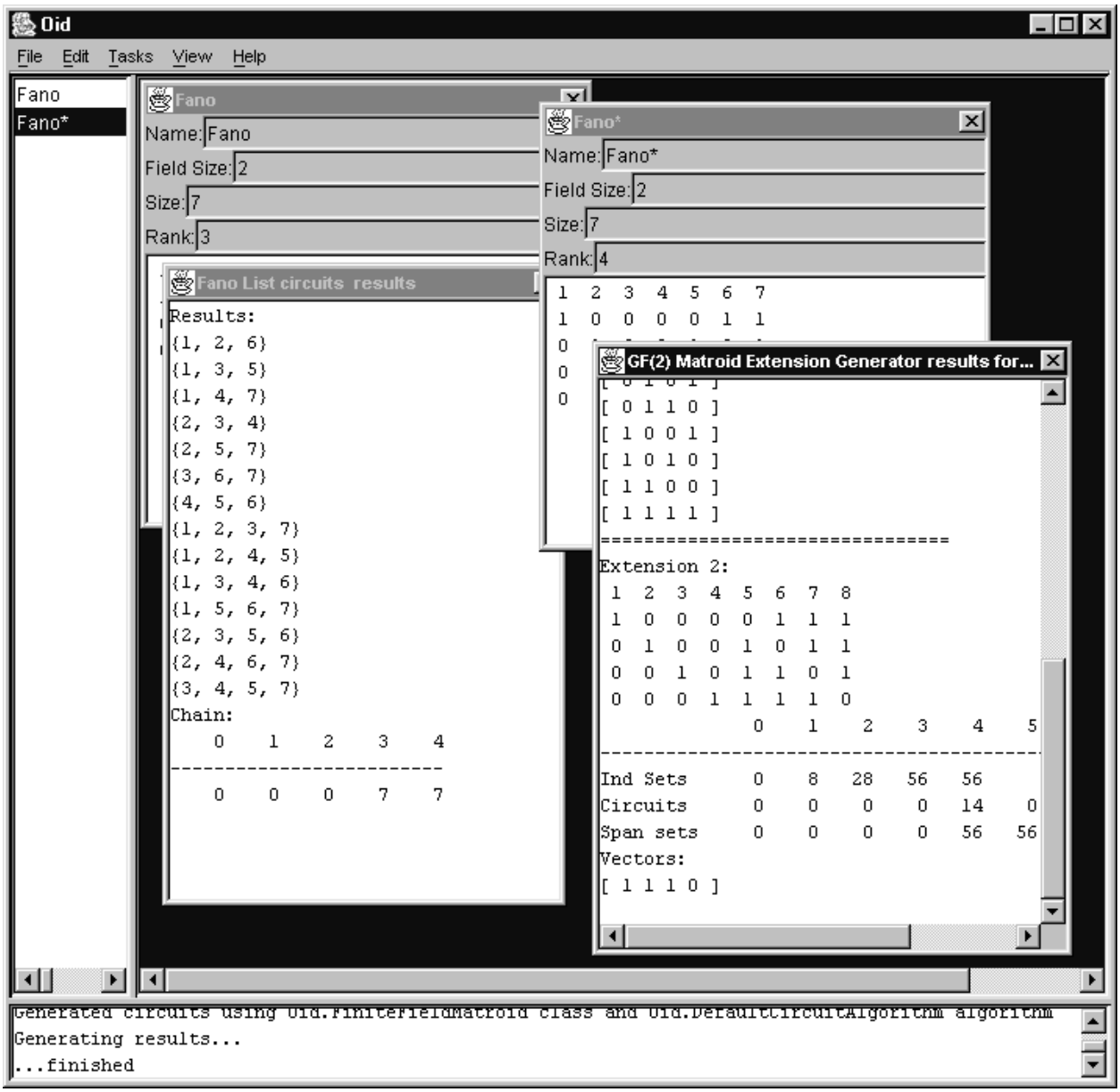


Figure 1.1: Oid's main window

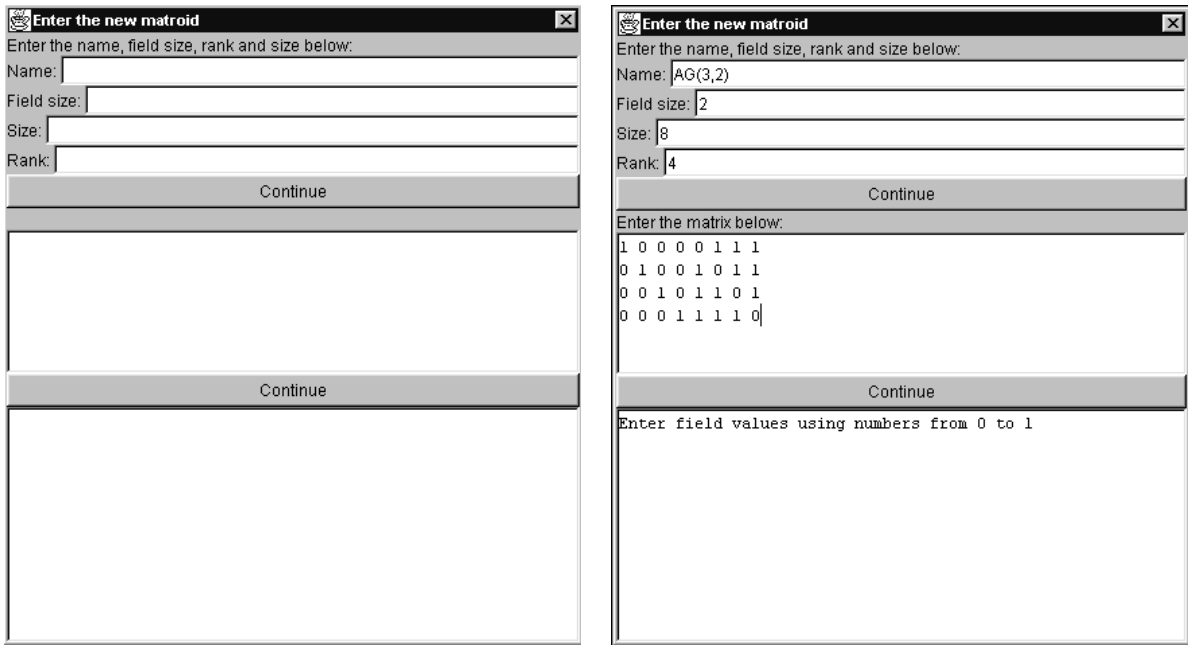
window, and its name will appear in the list box. The rank of the matroid must be at least 2 and the size of the matrix (number of columns) must be at least as large as the rank. If the rank and size of the matrix does not correpsond to the rank and size you entered, Oid will not let you proceed until you enter correct values.

Note that the size of the field must be a power of a prime. If the size of the field is a prime p, enter the matrix using numbers between 0 and $p-1$, inclusive. If the size of the field is a power of a prime, a legend will appear at the bottom of the window listing the field's elements and the value to be entered for each. Since the nonzero elements of any finite field form a cyclic group under multiplication, you will enter field elements using 0 and 1 to represent themselves, and (k) to represent the k^{th} power of a generator of this cyclic group.

If you selected **Projective geometry**, a similar window will appear, but with options only for the matroid's name, rank and field. Enter these values and Oid will generate the projective geometry of the correct rank over the field. Note that the size of the projective space of rank r over $GF(q)$ is $\frac{q^r-1}{q-1}$. So for large fields and ranks the projective geometry can take a long time to generate. This selction is not a different matroid representation since the representation of a projective geometry is a matrix. However, if you want to work with a projective geometry, it is far more convenient to let Oid generate it for you, than typing in the matrix.

If you selected **Basis representation**, an entry window like the one in Figure 1.3 will appear. Enter the matroid's name and size (≥ 2), and then enter the matroid's bases, one per line, in the box below. Use the numbers 1 through n to represent matroid elements, where n is the size of the matroid. When finished, click the "Continue" button at the bottom of the window. Oid will check the list of bases you entered to ensure it satisfies the postulates for a basis matroid: the set of bases is non-empty; and if B_1 and B_2 are two basis sets and $x \in B_1 - B_2$, then there is an element $y \in B_2 - B_1$ such that $(B_1 - x) \cup y$ is also a basis set. If so, Oid will display the new matroid. Otherwise, it will display an error message so you can correct your entries.

Finally, if you selected **Linear space**, Oid will present you with an option to enter a linear space as a rank 3 matrix or a list of its non-trivial lines. If you select the matrix, Oid will display a matrix entry screen with fixed rank 3. If you select lines, Oid will display a window similar to the basis entry window. Enter the non-trivial lines using the numbers 1 to n. Note that a line is non-trivial if it has three or more points. You may enter the trivial lines if you want to, but there is no need except in the rare case that your matroid has no non-trivial lines. Oid can figure out the trivial lines (2-point lines) using the non-trivial lines. Oid will check the list of non-trivial lines you entered to ensure it satisfies the postulates for a linear space: any two points belong to exactly one line; and any line has at least two distinct points. If so, Oid will display the new matroid. Otherwise, it will display an error message so you can correct your entries.



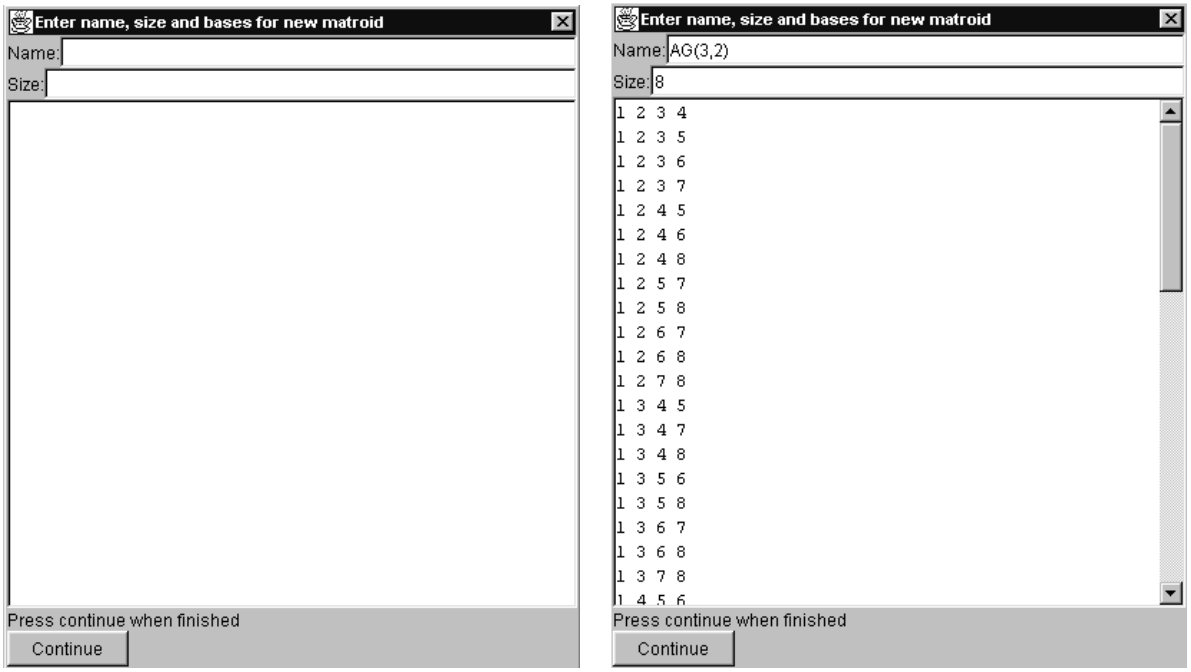
(a) Entry window (b) AG(3,2)

Figure 1.2: Entry window for a new finite field matroid

1.4 Saving and loading matroids

To save a matroid to a file, click "Save" on the File menu. Oid will prompt you to select a filename and location to save the matroid.

To load a previously saved matroid, click "Open" from the File menu. Oid will prompt you to locate a file. Select the matroid's file, and Oid will load the matroid and add it to the list of open matroids. Oid does not include any warning when exiting if you have not saved matroids. So be sure to save your work before exiting the program.



(a) Entry window (b) AG(3,2)

Figure 1.3: Entry window for a new basis matroid

Chapter 2

Matroid Tasks

Oid's matroid tasks are listed on the **Task menu** and **View menu**. We begin by describing each task on the Task menu in detail.

2.1 Task menu options

2.1.1 `Generate dual`, `Generate dual in standard form`

This task works for representable and non-representable matroids, but not for linear spaces. To generate the dual of a matroid you have loaded, select it from the list box on the left. Then select "Generate dual" from the Tasks menu. The dual of the matroid will appear and be added to the list of loaded matroids. If your matroid is represented as a matrix $[I_r|D]$, then its dual will appear as $\left[-D^T|I_{n-r}\right]$, where r and n are the rank and size of the matroid, respectively. If you wish to see the dual in standard form, with a rearrangement of the elements, select "Generate dual in standard form" from the Task menu.

2.1.2 `List circuits`, `List flats`, `List bases`, `List independent sets`, `List spanning sets`

These tasks work for representable and non-representable matroids, as well as, linear spaces. Each task generates a list of the requested subsets. For circuits, independent sets, bases, and spanning sets, the lists are followed by summary information based on the sizes of the subsets. At the end of the list you will see two rows of numbers. The numbers in the first row indicate the size of the subset. The numbers in the second row indicate the number of subsets of each size. The rank of a circuit is the size of the circuit minus 1; the rank of an

independent set is its size and the rank of a spanning set is the rank of the matroid. For flats, you will see two types of summary information - number of flats of different sizes and number of flats of different ranks. The latter gives the Whitney numbers of the second kind.

To get information on cocircuits, coindependent sets etc., first click on the "Generate dual" task and then click on the corresponding tasks for the dual matroid.

2.1.3 Circuit-cocircuit intersections

This task works for representable and non-representable matroids, as well as, linear spaces. It lists the circuit-cocircuit intersections of a matroid, organized by size, and a summary of the number of circuit-cocircuit intersections of each size.

2.1.4 Check two matroids for isomorphism

This task works for representable and non-representable matroids, as well as, linear spaces. Oid has an abstract isomorphism checker that can check for an isomorphism between two or more loaded matroids. You can compare matroids with different representations. For example, you can compare a $GF(q)$-representable matroid with a matroid represented by its family of bases.

Click on "Check two matroids for isomorphism" on the Task menu. Oid will display a list of the currently loaded matroids. Holding downs the Ctrl key on your keyboard, click on two of the matroids listed, and then click the "Ok" button. When the isomorphism check is complete, Oid will display a window with details of the check.

Oid's isomorphism checking routine first tries to rule out isomorphism by computing several matroid invariants such as the independent sets, circuits, and spanning sets. It determines the number of subsets of each size in each family. We call these sequences "chains". So we get the independent set chain, circuit chain, and spanning set chain. It also determines the number of sets in each family containing each element. We call these lists "degree sequences". So we get the independent set degree sequence, circuit degree sequence, and spanning set degree sequence. Most of the time, if two matroids are not isomorphic they will have different chain and degree sequences in one of the families. So it is frequently quick to determine if two matroids are non-isomorphic. The reader may be curious as to why we have used these three families. Selecting invariants to rule out isomorphism is largely an art. Through trial and error this gave us the fastest results. We could include flats, but the flats algorithm is not as fast as the circuit, independent set, and basis algorithms. We have a seperate isomorphism checking routine for rank 3 matroids which takes advantage of (non-trivial) flat chains and degree sequences as it is easy to get the flats of a rank 3 matroid. We could also include the dual chains, but for matroids with small rank and large

size, computation of dual chains would be time consuming. For various special operations we modify the isomorphism checker to include more invariants to speed up isomorphism testing.

If the two matroids have matching chains and degree sequences, then the isomorphism checker computes pseudo-orbits and begins enumerating mappings from one matroid to the other, subject to the restrictions imposed by the pseudo-orbits, and tests each mapping to see whether it preserves bases. This is also quite fast except in the case when the pseudo-orbits are large. One such instance is when the matroid has a transitive automorphism group. This is the worst case scenario where the isomorphism checker has to check potentially $n!$ maps. As soon as a mapping that preserves bases is found, the routine stops and outputs the map along with some process details.

2.1.5 `Check whether a binary matriod is graphic`

This task works for binary matroids. It uses Tutte's Algorithm[5] to a determine if a selected binary matrix is graphic. It outputs the results, along with details of the process.

2.1.6 `Display matroid polynomials`

This function outputs a table listing the number of matroid subsets of each size and rank, followed by the matroid's rank generating polynomial, chromatic polynomial, Tutte polynomial and connectivity polynomial. It also lists the matroid's chromatic number, critical number, its number of bases, independent sets and spanning sets, and its beta invariant (the coefficient on the x term in the matroid's Tutte's polynomial).

2.1.7 `Matroid connectivity function`

This task works for representable and non-representable matroids, as well as, linear spaces. The output lists each subset along with the value of the connectivity function for that subset. Specifically, it computes for each subset X, the value $\lambda(X) = r(X) + r(E - X) - r(M)$. The output can be used to determine the connectivity of the matroid. If M is a simple matroid and and for all X such that $min\{|X|, |E - X|\} \geq 3$, $\lambda(X) \geq 2$, then the matroid is 3-connected.

2.1.8 `Check matroid for negative correlation`

This task works for representable and non-representable matroids, as well as, linear spaces. It gives the size basis matrix and determines if the matroid is negatively correlated.

2.1.9 Generate a minor

This task works for $GF(q)$-representable matroids. It generates an arbitrary minor of a selected matroid. After you click the menu option, Oid will display a dialog box asking for elements to delete and contract. Enter a list of element labels in the form $d_1\ d_2 \ldots d_k/c_1\ c_2 \ldots c_k$, where the values d_i are the labels of elements to be deleted and the values c_j are labels of elements to be contracted. Oid will perform the operation and add the new minor to the list of loaded matroids. Note that if you want to only contract elements, then type $/c_1\ c_2 \ldots c_k$.

2.1.10 List modular cuts of a rank 3 matroid

This task works only for rank 3 matroids, entered as a matrix or as a linear space. It lists the matroid's non-isomorphic modular cuts. This gives us all the non-isomorphic single-element extensions of the matroid.

2.1.11 Single-element extensions of a GF(q) matroid

This task works for $GF(q)$-representable matroids. Clicking this task gives all the non-isomorphic $GF(q)$-representable single-element extensions grouped by isomorphism classes. For each extension it also gives a summary of the number of independent sets, circuits, and spanning sets of each size. This acts as a signature for the extension and is useful in keeping track of extensions.

Since the Splitter Theorem tells us that every 3-connected matroid can be obtained from its 3-connected minor by a sequence of single-element extensions and coextensions, this task is very useful for determining the structure of $GF(q)$-representable amtroids. More information on how to use this task is given in the next section.

2.1.12 Graphic/Cographic single-element extensions of graphic matroids

This task works for graphic matroids (graphs) represented in binary matroid form. It combines the single-element extension algorithm with Tutte's algorithm to determine the graphic and cographic single-element extensions of a graph. Note that, if an extension is both graphic and cographic, then it is planar.

2.1.13 Single-element deletions of a GF(q) matroid

This task works for $GF(q)$-representable matroids. It outputs the single-element deletions of a matroid in standard form.

2.1.14 Single-element contractions of a GF(q) matroid

This task works for $GF(q)$-representable matroids. It outputs the single-element contractions of a matroid in standard form.

2.1.15 Checks single-element deletions and contractions with Tutte's algorithm

This task works for $GF(q)$-representable matroids. It combines the deletion and contraction algorithms with Tutte's algorithm and checks whether each deletion and contraction is graphic or cogrpahic. If it is both graphic and cographic then it is planar.

2.2 View menu

2.2.1 Geometric representation

This task works for rank 3 matroids entered as a matrix or as a linear space. The code for this task was written by Padma Gunturi as part of her Master thesis [ ]. It displays the matroid's geometric representation in an interactive window. Select the matroid you want to display and click "Geometric representation" from the View menu to get a drawing of the points and lines. The points can be moved around the screen and the line structure will remain. This way you can get get different looking drawings of the same matroid. This is useful to get an insight into what it means for two geometric representations to be isomorphic and helps to build geometric intuition of matroids. The lines are in different colors to distinguish between them. The lines are Bezier curves, so if you toggle the control points button you will see the small control points on screen. You can move the control points to obtain curves. The geometric representation window also allows you to perform the following operations and see the effects visually:

1. Highlight a point or line
2. Delete or contract points and lines by selecting them from the list of flats to the left.

3. Relax a circuit hyperplane.
4. Obtain the bipartite graph (lattice of flats structure)
5. Obtain the $GF(q)$-complement of a $GF(q)$-representable rank 3 matroid.
6. Capture a particular drawing you like or reset the drawing to the original

Chapter 3

Oid Concepts

Oid is designed according to the principles outlined in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides [G]. Gamma *et al* define a *framework* as a set of cooperating classes that make up a reusable design for a specific class of software [G, p. 26]. Oid is designed to be a framework for systems that work with specific objects and treat algorithms for those objects as data. The framework has an extensible library of combinatorial object classes and utility classes that can be used to build a software system for other combinatorial objects. The library of object classes range from simple ones such as a subset class with subset operations to more complicated ones such as finite field classes.

3.1 Matroid representations

Since Oid is an extensible system, we can add other matroid representations such as circuit representations, flats representation etc. We are open to suggestions from users as to what representations they would find useful. Matroids and their different representations are like topological spaces and their different but equivalent definitions. A software system that seemlessly handles several different representations of matroids may act in some small way as a unifying concept for the cryptomorphic definitions and the seemingly different areas of matroids they give rise to.

Oid models different representations of matroids using Java interfaces. A matroid may be entered and stored in more than one way. For example, it can be entered as a matrix over a finite field or as a family of bases. Each such representation is modeled by a class that implements the object's interface.

In particular, Oid includes a Java interface called `Matroid`. Every class that implements

this interface must provide methods for certain common matroid functions, including the subset rank function and functions to determine whether a subset of the matroid is a circuit, independent set, basis, hyperplane or flat. Oid currently includes three different classes that implement this interface:

- The class `FiniteFieldMatroid` stores a matroid as represented by a set of columns over a finite field $GF(q)$. It determines subset rank using a routine that performs Gaussian reduction over $GF(q)$, and defines most of its other functions in terms of the rank function.
- The class `LinearSpace` stores a matroid as a set of lines. This representation is used heavily in the geometric representation viewer.
- The class `BasisMatroid` stores a matroid as a collection of bases.

3.2 Finite fields

Oid uses Java's object-oriented programming features in its classes for finite field arithmetic. It includes classes that are specific to fields $GF(q)$ where q is prime, where q is a power of 2, and where q is a power of another prime. A factory class, `FiniteFieldFactory`, provides a static method that, given a prime power q, returns an instance of a class implementing the `FiniteField` interface, but optimized internally for the specific value q.

Oid stores finite field elements as integers. When q is prime, operations over $GF(q)$ can be performed simply by reducing results modulo q, and by maintaining a table of the multiplicative inverses of all field elements that is computed when the field object is first created. For values of q that are prime powers, say $q = p^k$ where $k > 1$, the program uses the algorithm given in [4] to determine a primitive element ω of the field; that is, a cyclic generator of the field's multiplicative group of nonzero elements. Then, to determine the field element x from a stored integer m, m is expressed as a sum of powers of the field characteristic p: $m = a_0p^0 + a_1p^1 + \ldots + a_{k-1}p^{k-1}$. The coefficients a_i are then taken to be the coefficients on powers of ω to express the field element: $x = a_0\omega^0 + a_1\omega^1 + \ldots + a_{k-1}\omega^{k-1}$. When the field object is created, Oid determines the primitive element and computes tables for addition, negation, multiplication and multiplicative inverses. This scheme allows Oid to work with any finite field $GF(q)$ of a size q that will allow all field elements to be represented as Java integers.

3.3 Useful Oid classes

The Java classes described in this section are useful for creating new algorithms to be used in Oid itself, or they can be used to develop standalone programs. This section assumes general knowledge of Java programming, including object classes and Java interfaces. More information on Java programming may be found in [1].

Consider the Java program in Figure 3.1. This program creates an instance of the Fano matroid, represented as a matrix over $GF(2)$, uses an Oid algorithm to generate a list of its circuits, and prints the list of circuits to standard output.

3.3.1 Finite fields

To create an instance of a representable matroid, Oid needs an array of vectors of field elements. The variable `v` in the program above is an array of seven `FiniteFieldVector` references. `FiniteFieldVector` implements basic operations for vectors over finite fields. Next, we need a reference to the finite field we will use. The variable `GF2` is declared to be of type `FiniteField`. This is a Java interface in Oid implemented by three classes: `GFpCalculator`, which handles arithmetic in fields of the form $GF(p)$, p prime, `GF2kCalculator`, which handles arithmetic in fields of characteristic 2, and `GFpkCalculator`, which handles arithmetic in arbitrary finite fields (subject to storage limitations). Given a prime power q, the static method `CreateFiniteField` in the `FiniteFieldFactory` class selects the optimal implementation of `FiniteField` and returns an instance. In this program, `GF2` will be of type `GFpCalculator`, but we will refer to it only as an instance of `FiniteField`.

The next few lines after the `GF2` declaration create an array of integers holding a matrix representation of the Fano, and set up the array `v` of individual columns.

3.3.2 Matroid representations

Once we have an array of `FiniteFieldVector` references, the next step is to create an instance (`fano`) of `FiniteFieldMatroid`.

One of the key design strategies for Oid was flexibility with regard to matroid representations. We wished to design a system that would allow different representations of matroids to be used interchangably wherever possible. To do this, any Java class that models a particular matroid representation must implement the `Matroid` interface. If a class implements this interface, instances of the class can be passed to most of the Oid matroid algorithms without any modification. In particular, here `FiniteFieldMatroid` implements the `Matroid` interface.

```
import java.util.*;
import java.io.*;
import Oid.*;

public class CircuitDemo {
  /**
   * Computes and outputs the circuits of the Fano matroid.
   */
  public static void main(String[] arg) {
    FiniteFieldVector[] v = new FiniteFieldVector[7];
    int i,j;
    FiniteField GF2 = FiniteFieldFactory.createFiniteField(2);
    int[][] base
      = new int[][] {{1,0,0},{0,1,0},{0,0,1},
                     {0,1,1},{1,0,1},{1,1,0},{1,1,1}};
    for (i=0; i<7; i++) {
      v[i] = new FiniteFieldVector(base[i],3,GF2);
    }

    FiniteFieldMatroid fano=new FiniteFieldMatroid(v);

    DefaultCircuitAlgorithm cctAlg = new DefaultCircuitAlgorithm();

    cctAlg.setInstance(fano);
    Collection ccts = cctAlg.getSubsetList();

    for (Iterator p=ccts.iterator(); p.hasNext(); ) {
      System.out.println(p.next());
    }
  }
}
```

Figure 3.1: Program that lists the circuits in the Fano matroid

The `Matroid` interface

Classes implementing the `Matroid` interface must implement the following methods:

- `getName` and `setName` allow the matroid to be assigned a name.
- `getLabel` and `setLabel` allow individual elements of the matroid to be named, although these names are not used explicitly by Oid.
- `getRank` and `getSize` must return the matroid's rank and size, respectively.
- A number of methods check matroid subset properties: `isBasis`, `isCircuit`, `isFlat`, `isHyperplane` and `isIndependent`. `isInClosure` checks whether an element lies in the closure of a subset of the matroid's base set, and `closure` returns the closure of a set in the matroid.
- `subsetRank` evaluates the rank of a matroid subset.
- Finally, `getState` and `setState` are designed to allow the matroid to be stored and retrieved.

Oid also includes two other classes that implement `Matroid`. `BasisMatroid` stores a set of matroid bases, and determines matroid properties in terms of the bases. And an instance of `LinearSpace` stores the set of lines of a matroid and uses them to determine matroid properties.

Oid includes a few utility classes that are useful for handling `FiniteFieldMatroid`s:

- `FiniteFieldMatroidFormatter` produces strings displaying `FiniteFieldMatroid`s at several different levels of detail.
- `MatrixFileReader` loads a file containing matrices over finite fields and provides them to the calling program as `FiniteFieldMatroid` references.
- `PGFactory`, given a prime power q and a rank r, produces the projective geometry $PG(r-1, q)$ as an array of `FiniteFieldVector` references.

3.3.3 Matroid algorithms

Returning to the program in Figure 3.1, we are ready to pass our matroid instance to an algorithm to determine its circuits. The `DefaultCircuitAlgorithm` class accepts any `Matroid` and provides a list of its circuits. This list can also be organized by size, but we have not used that option here. The algorithm returns the list of circuits as a Java `Collection`, containing references to Oid `Subset` objects. Since `Subset` includes an overloaded `toString` method, we can output the list of circuits simply by iterating through the list and printing each circuit to standard output.

Several other Oid algorithms work the same way–in fact, exactly the same way, as they all implement a Java interface called `SubsetLister` for algorithms which produce lists of subsets as output. These include `DefaultBasisAlgorithm`, `FlatAlgorithm` (an improvement over the older `DefaultFlatAlgorithm`), `DefaultIndependentSetAlgorithm` and `SpanningSetAlgorithm`. Each of these algorithms can be used with any `Matroid` implementation.

3.3.4 Subsets and subset enumeration

In order to efficiently handle subsets of matroid base sets, Oid includes a family of classes for manipulating subsets. An instance of the class `Subset` stores a subset of a larger base set, internally using an instance of `java.util.BitSet` to represent the subset entries. Oid also includes a class `SubsetEnumeration` that, created with values n and k, enumerates all of the k-element subsets of the set $\{1, \ldots, n\}$, using the algorithm in *reference to Wilf.* `SubsetEnumeration` implements the Java interface `java.util.Enumeration`, making it very convenient to use in `for` loops. The method in Figure 3.2 determines which 3-element subsets of a matroid's ground set are circuits and lists them.

```
public void threeCcts(Matroid M) {
  // code to test 3-element subsets matroid ground set
  int n=M.getSize();
  Subset s;
  SubsetEnumeration enum;
  for (enum=new SubsetEnumeration(n,3); enum.hasMoreElements(); ) {
    s = (Subset)enum.nextElement();
    if (M.isCircuit(s)) {
      System.out.println(s);
    }
  }
}
```

Figure 3.2: Using `SubsetEnumeration` to test matroid subsets

Chapter 4

Adding new algorithms to Oid

Algorithms in Oid are implemented by classes that in turn implement generic algorithm interfaces. At runtime, a simple intelligent system delegates tasks to the appropriate algorithm. So, algorithms can be regarded as plug-ins to the system and new algorithms can be added without recompiling the existing code.

This chapter lists the steps necessary to add a new algorithm to the Oid program. It illustrates the process with an example: adding the `SpanningSetAlgorithm` to Oid.

4.1 Types of algorithms

Each algorithm class must specifiy which representations it works with and must implement one of the five algorithm interfaces listed below. These five interfaces are not meant to be exhaustive, rather just a reflection of our limited knowledge of what would be useful.

1. `SubsetLister` interface: Algorithms that implement this interface accept a matroid as input, and produce a collection of subsets as output. Examples include algorithms that list circuits, independent sets, and flats.

2. `PropertyCalculator` interface: Algorithms that implement this interface accept a matroid as input and produce a numerical value.

3. `SingleConstructor` interface: Algorithms that implement this interface accept a matroid as input and return another matroid as output. Examples include an algorithm which contracts a specified element from a matroid.

4. `CollectionConstructor` interface: Algorithms that implement this interface accept a matroid as input, and generate a collection of matroids as output. Examples include algorithms to generate all single-element contractions of a single matroid.

5. `Combiner` interface: Algorithms that implement this interface accept a collection of matroids as input, and generate a single matroid as output.

4.2 Creating a new algorithm

A new algorithm may be added to the Oid without recompiling any of the rest of the system's code. The steps are listed below:

1. Choose the right algorithm interface from the five mentioned above, and write the algorithm to implement the interface. It is easiest to begin with the skeleton provided by the interface, and then add code particular to the algorithm.

2. Specify which representations the algorithm works for. It should generate an exception if not given the correct representation.

3. Write methods to estimate the complexity of the algorithm. The algorithm should implement the `ComplexityProvider` interface. The algorithm's complexity estimating methods may, in turn, rely on its input representation's complexity estimating methods. This enables Oid to determine, for example, whether using a `FiniteFieldMatroid` or a `BasisMatroid` representation of a given matroid will result in faster execution.

4. The system reads a text file, `matroidTaskAlgorithmList.txt`, to determine which algorithms are available. When more than one algorithm is available to perform a specific task, each must implement the same algorithm interface. If the algorithm performs a new task, that is not already listed in the Tasks menu, add a line to `matroidTaskAlgorithmList.txt` describing the task as follows:

   `task` *taskName interfaceName taskDescription*

   Here, *interfaceName* refers to the algorithm interface the new algorithm implements, and *taskDescription* refers to the description to be displayed on the Tasks menu.

5. Add another line describing the new algorithm as follows:

   `algorithm` *className taskName representationClass algorithmDescription*

Here, *className* is the name of the Java class containing the algorithm, *taskName* is the same as in Step 4, *representationClass* is the matroid representation accepted by the algorithm as input, and *algorithmDescription* is the menu description for the algorithm. This description will appear only if there is more than one non-trivial algorithm for a specific task, in which case the system provides the user with a choice of algorithms on the Tasks menu.

6. Compile only the new algorithm class and run the system. The new algorithm should appear on the Tasks menu.

4.3 Example

In this section we will go through the steps necessary to add the `SpanningSetAlgorithm` to Oid.

4.3.1 Implementing `SubsetLister`

Since this algorithm lists the members of a family of sets, the appropriate algorithm interface is `SubsetLister`. We can use a list of the methods in this interface as a template to get started. Figure 4.1 lists the member functions of `SubsetLister`.

The methods `getName` and `getAlgorithmDescription` are self-explanatory and can be written easily for `SpanningSetAlgorithm`:

```
public String getName()
{
  return new String("Spanning Set Algorithm");
}


public String getAlgorithmDescription()
{
  String answer = new String("");
  answer ="Generates a list of the matroid's spanning sets.";
  return answer;
}
```

`getArrayIndexName` returns a string containing the name of the indexing value for the subsets returned by the algorithm. In this case, the sets will be indexed by size, so the method is as follows:

```
public interface SubsetLister {
  public void setInstance(Object arg)
    throws IllegalArgumentException;

  public Object getInstance();

  public Collection getSubsetList()
    throws NullPointerException;

  public Vector getSubsetListVector()
    throws NullPointerException;

  public Vector getSubsetCountChain()
    throws NullPointerException;

  public String getArrayIndexName()
    throws NullPointerException;

  public String getAlgorithmDescription();

  public String getName();
}
```

Figure 4.1: `SubsetLister` interface methods

```
public String getArrayIndexName()
  throws NullPointerException
{
  return new String("Size");
}
```

The rest of the methods form the heart of the algorithm. We will start with `setInstance`. This method allows the calling program to send a matroid to the algorithm for computation. Since the method only specifies that the passed item is a Java `Object`, we will first need to check to make sure it is a matroid:

```
public void setInstance(Object arg)
  throws IllegalArgumentException
{
  if (!(arg instanceof Matroid)) {
    throw new IllegalArgumentException("SpanningSetAlgorithm requires a "
                                       + "Matroid implementation");
  }
}
```

Next, we'll add data members to the class to store the matroid in use and the subset collections that the algorithm will compute. We'll also add a boolean data member to indicate whether the matroid is implemented as a `BasisMatroid`; the algorithm can proceed more quickly if it does not need to first compute the matroid's bases:

```
private Matroid mMatroid;
private boolean mIsBasisMatroid;
private Vector  mSpanningSets;
private Vector  mChain;
```

(Note: To use the Java `Vector` class, we will also need to import it from the `java.util` package.)

Now we can complete the `setInstance` method, by getting a local copy of the matroid and calling private methods to do the work.

```
public void setInstance(Object arg)
  throws IllegalArgumentException
{
  if (!(arg instanceof Matroid)) {
    throw new IllegalArgumentException("SpanningSetAlgorithm requires a "
```

```
                                        + "Matroid implementation");
  }

  mMatroid = (Matroid)arg;
  mIsBasisMatroid = (mMatroid instanceof BasisMatroid);
  determineSpanningSets();
  determineChain();
}
```

Here is the `determineSpanningSets` method:

```
private void determineSpanningSets()
{
  int r = mMatroid.getRank(), n = mMatroid.getSize(), k;
  SubsetEnumeration se;
  Iterator pBases;
  Subset s;
  Vector bases;
  boolean found;

  // first we need a Vector with all the matroid's bases -- if the
  // matroid is a BasisMatroid, just use the getBases method; otherwise,
  // create an instance of DefaultBasisAlgorithm and use that
  if (mIsBasisMatroid) {
    bases = ((BasisMatroid)mMatroid).getBases();
  }
  else {
    DefaultBasisAlgorithm basisAlg = new DefaultBasisAlgorithm();
    basisAlg.setInstance(mMatroid);
    bases = new Vector(basisAlg.getSubsetList());
  }
  // set up the vector that will hold the vectors of subsets
  // all the entries for 0 .. r-1 will be empty
  mSpanningSets = new Vector();
  for (k=0; k<r; k++) {
    mSpanningSets.add(k, new Vector());
  }
  mSpanningSets.add(r, bases); // r-th entry is the bases
  for (k=r+1; k<=n; k++) {
    mSpanningSets.add(k, new Vector());
```

```
    }

    // now loop through all the subset sizes greater than r
    try {
      for (k=r+1; k<=n; k++) {
        for (se = new SubsetEnumeration(n, k); se.hasMoreElements(); ) {
          s = (Subset)se.nextElement();
          // find out if there is a basis that contains this subset
          found = false;
          for (pBases = bases.iterator(); pBases.hasNext(); ) {
            if (s.subsetContains((Subset)pBases.next())) {
              found = true;
              break;
            }
          }
          // if so, add it to the correct vector in the list of vectors
          if (found) {
            ((Vector)mSpanningSets.get(k)).add(s);
          }
        }
      }
    }
    catch (Exception e) {
      // this "can't happen"
      throw new InternalError("Internal error in SpanningSetAlgorithm: " +
                              e.toString());
    }
  }
```

It first determines a list of the matroid's bases. If the matroid is a `BasisMatroid`, the routine calls the `getBases` method. Otherwise, it creates uses an instance of `DefaultBasisAlgorithm` to get the matroid's bases.

The spanning sets are organized by size using multiple Java `Vector` instances: for each size, the routine creates a `Vector`, and organizes these into a single `Vector` that is indexed by size. Then, for each size $k > rank(M)$, an instance of `SubsetEnumerator` is created to generate all of the k element subsets of the base set $\{1, \ldots, n\}$, where n is the size of the matroid. Each subset is checked against each basis. Any subset containing a basis is added to the `Vector` for its size.

The remaining methods are fairly straightforward to write. The method `determineChain` builds a `Vector` containing the sizes of each of the collections of spanning sets by size.

```
private void determineChain()
{
  mChain = new Vector();
  int k, n = mMatroid.getSize();
  for (k=0; k<n; k++) {
    mChain.add(new Integer(((Vector)mSpanningSets.get(k)).size()));
  }
}
```

The method `getSubsetListVector` returns the collection of spanning sets as they are generated by `determineSpanningSets`. The method `getSubsetCountChain` returns the `Vector` generated by
`determineChain`. The method `getSubsetList` builds a single `Vector` of `Subset` references containing all of the spanning sets and returns it:

```
public Collection getSubsetList()
  throws NullPointerException
{
  Vector answer = (Vector)null;
  if (mSpanningSets instanceof Vector) {
    answer = new Vector();
    for (Iterator pSS = mSpanningSets.iterator(); pSS.hasNext(); ) {
      answer.addAll((Vector)pSS.next());
    }
  }
  return answer;
}
```

4.3.2 Adding complexity estimates

Writing these methods gives us a class that meets the minimum requirements for an Oid algorithm. However, in order to be used in the Oid user interface, the algorithm also needs to be able to provide estimates of the complexity of itself, both in the abstract and with a given matroid. This allows Oid to choose between different algorithms when executing a task. These methods are listed in the `ComplexityProvider` interface, shown in Figure 4.2.

The argument `methodName` refers to the method the complexity is requested for, and in practice is either `getSubsetList`, `getSubsetListVector` or `getSubsetCountChain`. A call with any of these methods needs the complexity of generating the spanning sets to be returned. If `argument` is present it will contain the matroid the complexity estimate is being

```
public interface ComplexityProvider
{
  public long getComplexityEstimate(String methodName);
  public long getComplexityEstimate(String methodName, Object argument);
  public long getComplexityEstimate(String methodName, Collection arguments);
}
```

Figure 4.2: `ComplexityProvider` interface

computed for, so this version of the complexity estimation routine can use the matroid's size and rank. If `arguments` is present it will contain a collection of matroids; the total complexity of applying the algorithm to each matroid is required.

We will omit the details of implementing these methods. The finished object class is listed below.

4.3.3 Adding the algorithm to Oid

After the object class is finished, we compile it and place the resulting class file in the same folder as the other Oid classes. To make the class available to Oid, we edit the parameter file `matroidTaskAlgorithmList.txt` to include the task performed by the algorithm:

```
task getSpanningSets Oid.SubsetLister List spanning sets
```

This item contains the text that will appear on the Oid task menu. Then we add a line indicating that our new algorithm performs this task:

```
algorithm SpanningSetAlgorithm getSpanningSets Oid.Matroid Lists spanning sets by size
```

These changes, plus writing the algorithm class itself, are the only steps required to add the algorithm to Oid. No change is necessary to any of the existing Oid code.

4.3.4 Listing of `SpanningSetAlgorithm`

The finished code of `SpanningSetAlgorithm`, including comments, is as follows:

```
import java.util.*;
import java.math.*;

/**
 * SpanningSetAlgorithm implements the SubsetLister interface, and generates
 * the spanning sets of a matroid, organized by size.  It works for any
```

```
 * Matroid representation, but can take advantage of a BasisMatroid if it
 * is given one.
 *
 * It works by generating a list of the matroid's bases (either trivially
 * in the case of a BasisMatroid or using DefaultBasisAlgorithm), and then
 * iterating through all subsets of the matroid which are larger than
 * the rank, checking to see if each one contains a basis.  The subsets
 * which do contain a basis are kept.
 *
 * @author Robert J. Kingan and Sandra R. Kingan
 * @date 14 Nov 2001
 * @version 2002-08-26
 */
public class SpanningSetAlgorithm implements SubsetLister,
                                              ComplexityProvider
{
  //
  // private data members
  //
  private Matroid mMatroid;
  private boolean mIsBasisMatroid;
  private Vector  mSpanningSets;
  private Vector  mChain;

  //
  // SubsetLister interface methods
  //

  /**
   * sets the reference object for this SubsetLister
   *
   * @param arg reference object
   * @throws IllegalArgumentException if <code>arg</code> is not a
   * <code>Matroid</code>
   */
  public void setInstance(Object arg)
    throws IllegalArgumentException
  {
    if (!(arg instanceof Matroid)) {
      throw new IllegalArgumentException("SpanningSetAlgorithm requires a "
```

```
                                        + "Matroid implementation");
  }

  mMatroid = (Matroid)arg;
  mIsBasisMatroid = (mMatroid instanceof BasisMatroid);
  determineSpanningSets();
  determineChain();
}

/**
 * returns the reference object for this SubsetLister
 *
 * @return reference object
 */
public Object getInstance()
{
  Object answer = null;
  if (mMatroid instanceof Matroid) {
    answer = mMatroid;
  }
  return answer;
}

/**
 * generates a list of the relevant Subsets for the reference object
 *
 * @return <code>Collection</code> of subsets
 * @throws NullPointerException if matroid has not been set
 */
public Collection getSubsetList()
  throws NullPointerException
{
  Vector answer = (Vector)null;
  if (mSpanningSets instanceof Vector) {
    answer = new Vector();
    for (Iterator pSS = mSpanningSets.iterator(); pSS.hasNext(); ) {
      answer.addAll((Vector)pSS.next());
    }
  }
  return answer;
```

```
}

/**
 * generates a Vector containing indexed collections of Subsets for
 * the reference object
 *
 * @return <code>Vector</code> of subset collections
 * @throws NullPointerException if matroid has not been set
 */
public Vector getSubsetListVector()
  throws NullPointerException
{
  Vector answer = (Vector)null;
  if (mSpanningSets instanceof Vector) {
    answer = mSpanningSets;
  }
  return answer;
}

/**
 * generates a Vector containing the number of Subsets at each index
 * for the reference object
 *
 * @return <code>Vector</code> of subset counts
 * @throws NullPointerException if matroid has not been set
 */
public Vector getSubsetCountChain()
  throws NullPointerException
{
  Vector answer = (Vector)null;
  if (mChain instanceof Vector) {
    answer = mChain;
  }
  return answer;
}

/**
 * returns the name of the property by which the subsets are indexed
 *
 * @return name of indexed property
```

```
 * @throws NullPointerException if matroid has not been set
 */
public String getArrayIndexName()
  throws NullPointerException
{
  return new String("Size");
}

/**
 * returns a description of the algorithm used to generate subsets
 *
 * @return algorithm description
 */
public String getAlgorithmDescription()
{
  String answer = new String("");
  answer +=
    "It works by generating a list of the matroid's bases (either trivially";
  answer +=
    "for a BasisMatroid or using DefaultBasisAlgorithm), and then";
  answer +=
    "iterating through all subsets of the matroid which are larger than ";
  answer +=
    "the rank, checking to see if each one contains a basis.  The subsets";
  answer +=
    "which do contain a basis are kept.";
  return answer;
}

/**
 * returns the name of the algorithm
 *
 * @return algorithm name
 */
public String getName()
{
  return new String("Spanning Set Algorithm");
}

/**
```

```
 * private method determines the spanning sets
 */
private void determineSpanningSets()
{
  int r = mMatroid.getRank(), n = mMatroid.getSize(), k;
  SubsetEnumeration se;
  Iterator pBases;
  Subset s;
  Vector bases;
  boolean found;

  // first we need a Vector with all the matroid's bases -- if the
  // matroid is a BasisMatroid, just use the getBases method; otherwise,
  // create an instance of DefaultBasisAlgorithm and use that
  if (mIsBasisMatroid) {
    bases = ((BasisMatroid)mMatroid).getBases();
  }
  else {
    DefaultBasisAlgorithm basisAlg = new DefaultBasisAlgorithm();
    basisAlg.setInstance(mMatroid);
    bases = new Vector(basisAlg.getSubsetList());
  }

  //
  // set up the vector that will hold the vectors of subsets
  // all the entries for 0 .. r-1 will be empty
  //
  mSpanningSets = new Vector();
  for (k=0; k<r; k++) {
    mSpanningSets.add(k, new Vector());
  }
  mSpanningSets.add(r, bases); // r-th entry is the bases
  for (k=r+1; k<=n; k++) {
    mSpanningSets.add(k, new Vector());
  }

  // now loop through all the subset sizes greater than r
  try {
    for (k=r+1; k<=n; k++) {
      for (se = new SubsetEnumeration(n, k); se.hasMoreElements(); ) {
```

```
        s = (Subset)se.nextElement();
        // find out if there is a basis that contains this subset
        found = false;
        for (pBases = bases.iterator(); pBases.hasNext(); ) {
          if (s.subsetContains((Subset)pBases.next())) {
            found = true;
            break;
          }
        }
        // if so, add it to the correct vector in the list of vectors
        if (found) {
          ((Vector)mSpanningSets.get(k)).add(s);
        }
      }
    }
  }
  catch (Exception e) {
    // this "can't happen"
    throw new InternalError("Internal error in SpanningSetAlgorithm: " +
                            e.toString());
  }
}

/**
 * private method determines the subset count chain
 */
private void determineChain()
{
  mChain = new Vector();
  int k, n = mMatroid.getSize();
  for (k=0; k<n; k++) {
    mChain.add(new Integer(((Vector)mSpanningSets.get(k)).size()));
  }
}

//
// ComplexityProvider interface methods
//
/**
 * returns a value estimating the number of steps required to
```

```
 * execute the method with the passed name, or -1 if the method cannot
 * currently be executed
 *
 * @param methodName name of method
 * @return complexity estimate for method
 */
public long getComplexityEstimate(String methodName)
{
  long answer = -1;

  // if there is a matroid for this representation, we can estimate
  // complexity
  if ((methodName.equals("getSubsetList") ||
       methodName.equals("getSubsetListVector") ||
       methodName.equals("getSubsetCountChain")) &&
      (mMatroid instanceof Matroid)) {
    answer = getComplexity(mMatroid);
  }

  return answer;
}

/**
 * returns a value estimating the number of steps required to
 * execute the method with the passed name and argument, or -1 if the
 * method cannot be executed with the passed argument
 *
 * @param methodName name of method
 * @param argument argument for method
 * @return complexity estimate for method
 */
public long getComplexityEstimate(String methodName, Object argument)
{
  long answer = -1;

  // if there is a matroid for this representation, we can estimate
  // complexity
  if ((methodName.equals("getSubsetList") ||
       methodName.equals("getSubsetListVector") ||
       methodName.equals("getSubsetCountChain")) &&
```

```
      (argument instanceof Matroid)) {
    answer = getComplexity((Matroid)argument);
  }

  return answer;
}

/**
 * returns a value estimating the number of steps required to
 * execute the method with the passed name and collection of arguments
 * (which must be placed in correct order for iteration), or -1 if the
 * method cannot be executed with the passed arguments
 *
 * @param methodName name of method
 * @param arguments arguments for method
 * @return complexity estimate for method
 */
public long getComplexityEstimate(String methodName, Collection arguments)
{
  long answer = -1;
  Object argument;

  if (arguments.size() > 0) {
    argument = arguments.iterator().next();

    // if there is a matroid for this representation, we can estimate
    // complexity
    if ((methodName.equals("getSubsetList") ||
         methodName.equals("getSubsetListVector") ||
         methodName.equals("getSubsetCountChain")) &&
        (argument instanceof Matroid)) {
      answer = getComplexity((Matroid)argument);
    }
  }

  return answer;
}

/**
 * private method actually estimates the complexity
```

```
 */
private long getComplexity(Matroid m)
{
  long answer = -1;
  int n = m.getSize();
  int r = m.getRank();
  long c = nCk(n,r);
  int k;

  if (m instanceof BasisMatroid) {
    answer = 1;
  }
  else {
    answer =
      ((ComplexityProvider)m).getComplexityEstimate("isIndependent") * c;
  }

  for (k=r+1; k<=n; k++) {
    answer += c * nCk(n, k);
  }

  return answer;
}

//
// private method to get combinations -- only uses BigInteger if
// it has to
//
private long nCk(long n, long k)
{
  long answer = 0;
  long i;

  if ((0 == k) || (n == k)) {
    answer = 1;
  }
  else if ((1 == k) || ((n-1) == k)) {
    answer = n;
  }
  else if ((k > 1) && (k < n-1)) {
```

```
      if (n > 20) {
        BigInteger bigAns = BigInteger.valueOf(1);
        for (i=n; i>n-k; i--) {
          bigAns = bigAns.multiply(BigInteger.valueOf((long)i));
        }
        for (i=n-k; i>1; i--) {
          bigAns = bigAns.divide(BigInteger.valueOf((long)i));
        }
        answer = bigAns.longValue();
      }
      else {
        answer = 1;
        for (i=n; i>n-k; i--) {
          answer *= i;
        }
        for (i=n-k; i>1; i--) {
          answer /= i;
        }
      }
    }

    return answer;
  }

}
```

Bibliography

[1] Ken Arnold and James Gosling. *The Java Programming Language, Second Edition*. Sun Microsystems, Mountain View, CA, 1998.

[2] R. J. Kingan and S. R. Kingan. A software system for matroids, graphs and discovery. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, (to appear).

[3] R. J. Kingan, S. R. Kingan, and Wendy Myrvold. On matroid generation. In *Proceedings of the Thirty-Fourth Southeastern International Conference on Combinatorics, Graph Theory, and Computing, Congressus Numerantium*, volume 164, pages 95–109, 2003.

[4] Sean E. O'Connor. Computing primitive polynomials: Theory and algorithm. http://www.seanerikoconnor.freeservers.com.

[5] W. T. Tutte. An algorithm for determining whether a given binary matroid is graphic. *Proceedings of the American Mathematical Society*, 11 (6):905–917, 1960.